悪条件連立一次方程式の精度保証付き数値計算法の研究

Studies on Numerical Verification Method for Ill-conditioned Simultaneous Linear Equations

2006年2月

早稲田大学大学院理工学研究科
情報・ネットワーク専攻　情報数理工学研究

太田 貴久

目 次

表 目 次

第1章　序論・準備

精度保証付き数値計算と言えば，昔は区間 $x=[\underline{x},\overline{x}]$，$y=[\underline{y},\overline{y}]$ に対して $x+y=[\underline{x}+\underline{y},\overline{x}+\overline{y}]$，$x-y=[\underline{x}-\overline{y},\overline{x}-\underline{y}]$，$x\times y=[\min\{\underline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\underline{y},\overline{x}\times\overline{y}\}$，$\max\{\underline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\underline{y},\overline{x}\times\overline{y}\}]$，$x/y=[\min\{\underline{x}/\underline{y},\underline{x}/\overline{y},\overline{x}/\underline{y},\overline{x}/\overline{y}\}$，$\max\{\underline{x}/\underline{y},\underline{x}/\overline{y},\ \overline{x}/\underline{y},\overline{x}/\overline{y}\}]$ (但し，$\underline{y}\le 0\le\overline{y}$ の時は定義しない) の様に区間演算を行っていたが，この原始的な方式をそのまま用いると数値解を求める計算に対して精度保証に何十倍から何百倍もの計算時間がかかってしまい，実用的ではなかった．しかし，近年の研究成果により，連立一次方程式の解の精度保証が解を求める計算時間とほぼ同じ時間で出来るようになり，精度保証が実用のレベルに達した．

本学位論文は，今までに作成してきた論文の集大成である．以下に，本論文の構成を示す．第 1 章では，本論文で用いる表記などの準備を行った．第 2 章では，高精度内積計算アルゴリズムを用いた連立一次方程式の精度保証付き数値計算法について述べた．また，高速性が損なわれずに高精度化されたことを数値実験で確かめた．しかし，高精度に計算してもこの方法だと条件数が 10^{16} 程度になると倍精度浮動小数点数演算での計算の限界であった．そこで，第 3 章では，条件数が 10^{16} 程度を超えても精度保証付き数値計算を可能にした．特に，第 3.4 節では，S. M. Rump 教授（ハンブルク工科大学）の任意に大きな条件数を設定できる乱数発生関数を使う代わりに条件数の指定は出来ないが大きな条件数を発生させる自作の乱数発生関数を使っていた時，解の絶対値の大小が大きく指数関数的にばらついたため山本の定理を使っても絶対値の小さな成分に対する精度保証がシャープに行えなくなったのでスケーリングを行うことによってこれを解決した．最後に，高精度精度保証アルゴリズムとしては，Ogita, Rump, Oishi による高速な任意精度内積演算アルゴリズムを用いて高速化に成功し，多倍長浮動小数点数計算を用いた場合と比べてかなり高速に計算できる事を数値実験で確かめた．

1.1　はじめに

本論文では，連立一次方程式

$$Ax=b \tag{1.1.1}$$

の数値解の精度保証付き数値計算法について考える．ただし，$n\times n$ 行列 A は実行列で，b は実 n 次元ベクトルとする．ここに，行列 A が実とは，その要素が全て実数であることをいう．以下，本論文では，記号として，$\mathbb{R}$ で実数の集合，$\mathbb{R}^n$ で実 n 次元ベクトル空間を表す．また，ベクトル $x\in\mathbb{R}^n$ の最大値ノルムを

$$\|x\|_\infty=\max_{1\le i\le n}|x_i| \tag{1.1.2}$$

で，行列 $A\in\mathbb{R}^{n\times n}$ の（最大値ノルムによる）作用素ノルムを

$$\|A\|_\infty=\sup_{\|x\|_\infty=1}\|Ax\|_\infty=\max_{1\le i\le n}\sum_{j=1}^{n}|A_{ij}| \tag{1.1.3}$$

で表す．x_i は x の第 i 成分で，A_{ij} は A の第 (i,j) 成分とする．

式 (1.1.1) の形の連立一次方程式は科学技術計算の各分野に現れることが知られている．特に，科学技術計算の分野での応用に際しては，問題の規模である次元 n の大きい問題が現れることが多い．そのような次元数が大きい場合，式 (1.1.1) の数値解は計算機ハードウェアに実装された浮動小数点演算により，計算されることが普通である[1]．本論文でも，行列 A とベクトル b の各要素は浮動小数点数とし，解の計算には浮動小数点演算を用いるという立場を取る．浮動小数点演算は計算機が発明されて以来，長い間標準化されなかったが，1975 年から 10 年間のデファクトスタンダードの期間を経て，1985 年に IEEE754 の 2 進浮動小数点数規格が制定され，今日ではその規格に従った CPU がほとんどとなっている．IEEE754 規格の中では，32 ビットの単精度浮動小数点数と 64 ビットの倍精度浮動小数点数およびそれらの演算が標準化されている．以下，本論文では IEEE754 規格に従う倍精度浮動小数点数での数値計算のみを考えることにするが，単精度浮動小数点数を用いる場合も同様の議論が成立する．

この IEEE754 倍精度浮動小数点数規格は数学的に美しい性質を備えたもので，精度保証付き数値計算に必要な方向丸めの概念も実装されている．これを利用して，Oishi と Rump [10] は，IEEE754 規格での内積計算の精度保証が 2 回の丸めモードの変更でできることを指摘し，それを基礎に丸めモード制御方式精度保証付き数値計算法を開発した．これにより，IEEE754 規格の倍精度浮動小数点演算のもとで，式 (1.1.1) の数値解を高速に精度保証できることも示されている．

従来，数値解の精度保証のための計算時間は，その数値解を求めるための計算時間に対して数百倍から数千倍程度かかるのが常識的であった．ここでの高速というのは，精度保証にかかる計算時間 $t_{\rm ver}$ が，数値解を求めるための計算時間 $t_{\rm sol}$ に比べて数倍程度であることをいうことにする．実際，文献 [10] では，A が密行列でガウスの消去法などの直接解法が倍精度演算の範囲で適用できるような問題のクラスに対しては，式 (1.1.1) において $t_{\rm ver}$ が $t_{\rm sol}$ の 5 倍程度（計算量は 8 倍）となる方法が示された．特に，問題の次元数 n が 5000 程度以下で係数行列が密行列かつ条件数がそれほど大きくない場合，$t_{\rm ver}$ と $t_{\rm sol}$ がほぼ同じくらいになる方法も示されており [10]，さらに文献 [16] では，密行列の中でより大規模な問題や条件数のある程度大きいクラスの問題に対しても，$t_{\rm ver}$ が $t_{\rm sol}$ の 3 倍程度（計算量は 4 倍）で済む方法が示されている．

1.2 浮動小数点演算と高精度内積計算

本論文で必要となる IEEE754 規格を満たす倍精度浮動小数点数に成り立つ重要な性質を概観する．

1.2.1 IEEE754 の倍精度浮動小数点数規格

まず，IEEE754 の倍精度浮動小数点数規格について概観しよう．IEEE754 規格での倍精度浮動小数点数は 64 ビットで表される浮動小数点数であり，つぎのような形をしている．

$$f = (-1)^s \times 2^e \times k \tag{1.2.1}$$

[1]問題の次元数が小さく，かつ，A と b の要素が有理数である場合には，有理数演算により式 (1.1.1) の正確な解を求めることができる．この場合には精度保証は必要なくなる．しかし，実際的な計算時間とメモリ量の範囲内で扱いうる問題の次元はかなり低く制限される．実際，通常の PC の場合には取り扱える次元の最大が数百次元程度と思われる．有理数演算では，この程度でも，式 (1.1.1) を解くのに一日かかる．また，その計算時間は浮動小数点数計算に比べて n の指数関数的オーダで増大する．

ここに，s は符号ビット（正ならば 0，負ならば 1），e は下駄履き表現された指数部（11 ビット），k は仮数部（53 ビット）である．e および k はそれぞれ，2 進数整数，2 進数小数である．仮数部は，正規化できる範囲では $k = 1.k_1k_2\cdots k_{52}$ のように先頭の 1 ビットを暗黙に仮定しているので，実際には 52 ビットを保持するだけで 53 ビットを表現している．f の値が大きすぎて正規化できない場合をオーバフロー，小さすぎて正規化できない場合をアンダフローと呼ぶが，これらの詳細については文献 [1] を参照されたい．

$\mathbb{F}$ で IEEE754 規格の倍精度浮動小数点数の集合を表すことにする． IEEE754 規格では倍精度浮動小数点演算は丸めモードを指定することにより定義されている．丸めモードは IEEE754 では 4 種類あるが，ここでは，本論文に関係する 3 つの丸めモードの定義を紹介する．

(1)$-\infty$ 方向への丸め（下への丸め） $c \in \mathbb{R}$ を $f \leq c$ を満たす最も大きな浮動小数点数 $f \in \mathbb{F}$ へ丸める．下への丸めを $\bigtriangledown : \mathbb{R} \to \mathbb{F}$ で表す.

(2)$+\infty$ 方向への丸め（上への丸め） $c \in \mathbb{R}$ を $f \geq c$ を満たす最も小さな浮動小数点数 $f \in \mathbb{F}$ へ丸める．これを $\bigtriangleup : \mathbb{R} \to \mathbb{F}$ と表す.

(3) 最近点への丸め $c \in \mathbb{R}$ を $|f - c|$ が最小となる浮動小数点数 $f \in \mathbb{F}$ へ丸める．これを $\square : \mathbb{R} \to \mathbb{F}$ と表す.

IEEE754 規格での倍精度浮動小数点演算は，四則演算子 $\cdot \in \{+, -, \times, /\}$ と丸め演算子 $\bigcirc \in \{\bigtriangleup, \bigtriangledown, \square\}$ に対して

$$x \bigodot y = \bigcirc(x \cdot y), \ (\forall x, y \in \mathbb{F}) \tag{1.2.2}$$

が満たされるように定義される．ここで，式 (1.2.2) の左辺の $\bigodot$ は浮動小数点演算を表し，$x \cdot y$ は通常の実数演算を表すものとする．

この定義から任意の $x, y \in \mathbb{F}$ と $\cdot \in \{+, -, \times, /\}$ に対して

$$\bigtriangledown(x \cdot y) \leqq x \cdot y \leqq \bigtriangleup(x \cdot y)$$

が成り立つ．したがって，特に，$x_i, y_i \in \mathbb{F}, \ (i = 1, 2, \cdots, n)$ のとき

$$fl_{\bigtriangledown}(\sum_{i=1}^{n} x_i \times y_i) \leqq \sum_{i=1}^{n} x_i \times y_i \leqq fl_{\bigtriangleup}(\sum_{i=1}^{n} x_i \times y_i)$$

が成り立つことがわかる．ここで，$fl_{\bigtriangleup}$ は $+\infty$ 方向の丸めモードで，$fl_{\bigtriangledown}$ は $-\infty$ 方向の丸めモードでそれぞれ浮動小数点演算を行うことを表す．これは内積の上限と下限が厳密に計算できることを示している．また，式 (1.2.1) から，倍精度浮動小数点数の絶対値が正しく計算できることや（$s = 0$ とすればよい），2 つの倍精度浮動小数点数の大小が正しく判定できることがわかる．したがって，$v \in \mathbb{F}^n$ が

$$\underline{v}_i \leqq v_i \leqq \overline{v}_i$$

を満たしているとき

$$\|v\|_\infty \leqq \max_{1 \leqq i \leqq n} (\max\{|\underline{v}_i|, |\overline{v}_i|\})$$

によって，v の最大値ノルムの上限が計算できることがわかる．同様に，$A = (a_{ij}) \in \mathbb{F}^{n \times n}$ で

$$\underline{a}_{i,j} \leqq a_{i,j} \leqq \overline{a}_{i,j}$$

となるとき

$$\|A\|_\infty \leqq \max_{1\leqq i\leqq n}\left[fl_\triangle\left(\sum_{j=1}^{n}\max\{|\underline{a}_{i,j}|,|\overline{a}_{i,j}|\}\right)\right]$$

によって A の最大値ノルムの上界が計算できることがわかる．

以上の事柄を基礎に，文献 [10] では式 (1.1.1) の高速精度保証の理論が展開されており，それは本論文での議論の基礎ともなっている．本論文での新しい提案の基礎はつぎの小節で紹介される．

1.2.2 高精度内積計算アルゴリズム

1950 年代の Runge-Kutta 法に対する Gill の丸め誤差対策の技法に関する議論から出発して，浮動小数点数の加算の誤差に対する深い議論がされるようになった．その大きな成果の一つが次の 1969 年の Knuth の定理である．ここで，定理中のアルゴリズムは可読性の点から MATLAB のプログラムに似た形式で表現しており，$\oplus$ 及び $\ominus$ は，それぞれ倍精度浮動小数点演算による加算と減算を意味する．また，特に断りのない限り最近点への丸めモードで演算は実行される．

定理 1 (Knuth [5]) $a,b\in\mathbb{F}$ とする．$x\in\mathbb{F}$ と $y\in\mathbb{F}$ をつぎのアルゴリズム **TwoSum** によって計算すると $a+b=x+y$ が成立する．

function $[x,y]=\mathbf{TwoSum}(a,b)$
$x=a\oplus b;$
$b_V=x\ominus a;\quad a_V=x\ominus b_V;\quad b_R=b\ominus b_V;\quad a_R=a\ominus a_V;$
$y=a_R\oplus b_R;$

（定理終）

定理 1 は和 $a+b$ を $a\oplus b$ で近似すると，その誤差 $a+b-(a\oplus b)$ が再び $\mathbb{F}$ の要素となり，$y\in\mathbb{F}$ として厳密に計算されるという驚くべき事実を示している．また，アルゴリズム **TwoSum** は単なる加減算の組み合わせで書かれており，条件分岐を含まないので，浮動小数点演算のステップ数は 6 ステップであっても，実際の数値計算においては，コンパイラの最適化のため，$x=a\oplus b$ を計算する時間に対して 6 倍もかからず，高速に計算される点に注意する．

1971 年，Dekker はこれと同様な定理が浮動小数点数の乗算についても成立することを示した．まず，準備として，次の定理を示す．

定理 2 (Dekker [2]) $a\in\mathbb{F}$ とする． つぎのアルゴリズム **Split** によって，仮数部の先頭ビットからの有効桁数（暗黙の 1 ビットを含む）がそれぞれ 26 ビット以内（27 ビットめ以降はすべてゼロ）である a_H と a_L を計算すると，$a=a_H+a_L$ で $|a_H|\geq|a_L|$ となり，*nonoverlapping* な和となる．ただし，$x,y\in\mathbb{F}$ が *nonoverlapping* であるとは，$x=r\times 2^s$ かつ $|y|<2^s$ あるいは $y=r\times 2^s$ かつ $|x|<2^s$ の様な整数 r と s が存在する事をいう．

function $[a_H,a_L]=\mathbf{Split}(a)$
$c=(2^{27}+1)\otimes a;\quad d=c\ominus a;$
$a_H=c\ominus d;$
$a_L=a\ominus a_H;$

（定理終）

この定理を基に，Dekker は次の定理が成り立つことを示した．定理中のアルゴリズムは，Veltkamp によるものである．

定理 3 (Dekker [2]) $a, b \in \mathbb{F}$ とする．$x \in \mathbb{F}$ と $y \in \mathbb{F}$ をつぎのアルゴリズム **TwoProduct** で計算すると， 計算の途中でアンダフローが起きなければ，$a \times b = x + y$ が成り立つ．

function $[x, y] = \mathbf{TwoProduct}(a, b)$
$x = a \otimes b;$
$[a_H, a_L] = \mathbf{Split}(a); \quad [b_H, b_L] = \mathbf{Split}(b);$
$c_1 = x \ominus a_H \otimes b_H; \quad c_2 = c_1 \ominus a_L \otimes b_H; \quad c_3 = c_2 \ominus a_H \otimes b_L;$
$y = a_L \otimes b_L \ominus c_3;$

（定理終）

アンダフローが起きた場合，著者らは次の結果を得ている．

定理 4 (Ogita-Rump-Oishi [9]) 定理 *3* においてアンダフローが計算途中で起きたとすると次の評価が成立する：

$$|x + y - a \times b| \leqq 4\eta \tag{1.2.3}$$

ただし，η はアンダフローユニット（*IEEE754* 倍精度のとき，$\eta = 2^{-1074}$）である． （定理終）

しかし，実際にはアンダフローが起きることはまれであることと，簡単のため，本論文ではアンダフローが起きた場合の処理については議論しないことにする．

以上の定理を基に，ベクトル $p = (p_1, p_2, \ldots, p_n)^T \in \mathbb{F}^n$ に対して，ベクトル $q = (q_1, q_2, \cdots, q_n)^T \in \mathbb{F}^n$ を計算するアルゴリズム **VecSum** を導入する．ただし，^{T} は転置を意味する．このとき，$q_n = fl(\sum_{i=1}^{n} p_i)$ となり，$q_1, q_2, \ldots, q_{n-1}$ は q_n の計算誤差を表す．

アルゴリズム 1 (Ogita-Rump-Oishi [9]) ベクトルの総和の無誤差変換法：

function $q = \mathbf{SumEFT}(p)$
$q = p;$
for $i = 2 : n$
$\quad [q_i, q_{i-1}] = \mathbf{TwoSum}(q_i, q_{i-1});$

（アルゴリズム終）

定理 1 より，$p \in \mathbb{F}^n$ を入力として，**SumEFT** で $q \in \mathbb{F}^n$ を計算したとすると，明らかに

$$\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i \tag{1.2.4}$$

が成り立つ[2]．

さらに，つぎの内積の高精度計算アルゴリズム **Dot2** を導入する．

[2] このような性質を持つアルゴリズムは “distillation algorithm” と呼ばれる．

アルゴリズム 2 (Ogita-Rump-Oishi [9]) 内積計算の高精度計算法：

$$
\begin{aligned}
&\textbf{function}\ \ s = \textbf{Dot2}(x, y)\\
&[p_1, s_1] = \textbf{TwoProduct}(x_1, y_1);\\
&for\ \ i = 2 : n\\
&\qquad [h_i, r_i] = \textbf{TwoProduct}(x_i, y_i);\\
&\qquad [p_i, q_i] = \textbf{TwoSum}(p_{i-1}, h_i);\\
&\qquad s_i = s_{i-1} \oplus (q_i \oplus r_i);\\
&s = p_n \oplus s_n;
\end{aligned}
$$

（アルゴリズム終）

この **Dot2** を用いると，$x, y \in \mathbb{F}^n$ に対して内積 $x^T y = \sum_{i=1}^{n} x_i \times y_i$ を倍精度演算のさらに 2 倍の精度，すなわち 4 倍精度で計算したかのような結果が得られることが示されている [9].

また，つぎの内積の無誤差変換アルゴリズム **DotEFT** を導入する.

アルゴリズム 3 内積計算の無誤差変換法：

$$
\begin{aligned}
&\textbf{function}\ \ v = \textbf{DotEFT}(x, y)\\
&[p_1, v_1] = \textbf{TwoProduct}(x_1, y_1);\\
&for\ \ i = 2 : n\\
&\qquad [h_i, v_i] = \textbf{TwoProduct}(x_i, y_i);\\
&\qquad [p_i, v_{n+i-1}] = \textbf{TwoSum}(p_{i-1}, h_i);\\
&v_{2n} = p_n;
\end{aligned}
$$

（アルゴリズム終）

この **DotEFT** を用いると，$x, y \in \mathbb{F}^n$ に対して

$$x^T y = \sum_{i=1}^{2n} v_i \tag{1.2.5}$$

という内積からベクトルの総和への誤差の無い変換が実行される．また，このようにして得られたベクトル v に対して **SumEFT** を繰り返し適用することによって，任意に精度の良い内積計算が可能となる.

1.2.3 高速精度保証法

連立一次方程式 (1.1.1) の高速精度保証では，つぎの定理が基礎となる[3].

定理 5 (Banach) A を $n \times n$ 実行列，b を n 次元実ベクトルとして連立一次方程式

$$Ax = b \tag{1.2.6}$$

を考える．適当な $n \times n$ 実行列 R が存在して

$$\|RA - I\|_\infty < 1 \tag{1.2.7}$$

[3] この定理は古くから知られており，実質的に Banach の縮小写像の原理が起源であると思われるので，Banach に負うものとする.

を満たすとき，A は可逆で，任意の n 次元実ベクトル $\tilde{x}$ に対して

$$\|x^* - \tilde{x}\|_\infty \leqq \frac{\|R(A\tilde{x} - b)\|_\infty}{1 - \|RA - I\|_\infty} \tag{1.2.8}$$

となる．ただし，I は n 次元単位行列で，$x^* = A^{-1}b$ とする． （定理終）

定理 5 において，実際の応用では R としては A の近似逆行列を取り，$Ax = b$ の近似解を $\tilde{x}$ とする．この定理を基に，連立一次方程式の精度保証法を実装する．

A を $n \times n$ 行列として連立一次方程式 $Ax = b$ を解くとき，$\mathrm{cond}(A) = \|A\|\|A^{-1}\|$ をその条件数という．ここで，b は n 次元ベクトルである．b が $b + \Delta b$ に変化したとき，解は $\mathrm{cond}(A)\Delta b$ のオーダーで変化する．したがって，条件数の大きな問題では，右辺ベクトルの少しの変化が解の大きな変化を引き起こす．IEEE754 の倍精度浮動小数点演算で計算する場合，仮数の精度は高々 $2^{-53} = 1.110\ldots \times 10^{-16}$ であるから，条件数が $\mathcal{O}(10^{16})$ に近くなると，丸め誤差の混入する数値計算での数値解の精度は仮数部の精度で 1 桁あるかないかというオーダーになり，計算の限界となる．ここでは，$\mathcal{O}(10^{14})$ 程度より大きな条件数を持つ問題を悪条件な問題と呼んでいる．ただし，本論文で扱う問題の範囲は A の条件数が $\mathcal{O}(10^{16})$ より小さい問題である．それよりも条件数が大きい問題に対する精度保証法については，文献 [15] 及び第 3 章を参照されたい．

第2章　高精度内積計算を用いた連立一次方程式の精度保証付き数値計算法

本章では，IEEE754 規格を満たす倍精度浮動小数点演算のみを用いて高速かつ高精度に内積計算を行うアルゴリズムを用いて，A が密行列でガウスの消去法などの直接解法が倍精度演算の範囲で適用できるような問題のクラスに対しては，その高精度内積計算法を利用して，式 (1.1.1) の数値解の残差反復計算が高速に実行できることを述べるとともに，その数値解の高速で高精度な精度保証法を提案する．本論文の議論のポイントは，このような高精度な精度保証法が，丸めモード制御方式精度保証付き数値計算法の高速性を損なうことなく実装できることを示すことである．

2.1　高精度精度保証法

2002 年，Oishi と Rump [10] は式 (1.1.1) の数値解に対する高速精度保証法を提案した．ここでは，この精度保証法を内積の高精度計算アルゴリズム **Dot2** 及び **DotEFT** を用いて高速かつ高精度に実装する方法を示す．

2.1.1　残差反復法

本節では，連立一次方程式 (1.1.1) の数値解に対する残差反復法について考える．ただし，$A \in \mathbb{F}^{n\times n}$ で $x, b \in \mathbb{F}^n$ とする．$\tilde{x} \in \mathbb{F}^n$ を式 (1.1.1) の数値解とする．$A_e \in \mathbb{F}^{n\times(n+1)}$ と $\tilde{x}_e \in \mathbb{F}^{n+1}$ を

$$A_e = (A|b), \quad \tilde{x}_e = \begin{pmatrix} \tilde{x} \\ -1 \end{pmatrix} \tag{2.1.1}$$

のように定義する．ただし，$-1 \in \mathbb{F}$ である．このとき

$$A_e\tilde{x}_e = A\tilde{x} - b$$

が成立する．ここで

$$\tilde{r} = (\mathbf{Dot2}(A_e^{(1)}, \tilde{x}_e), \mathbf{Dot2}(A_e^{(2)}, \tilde{x}_e), \ldots, \mathbf{Dot2}(A_e^{(n)}, \tilde{x}_e))^T \tag{2.1.2}$$

とする．ただし，$A_e^{(i)}$ は行列 A_e の第 i 行目を転置した列ベクトルとする．これにより，式 (2.1.2) の $\tilde{r}$ は残差 $r = A\tilde{x} - b$ を内積の高精度計算アルゴリズム **Dot2** によって計算したものとなる．次に，倍精度浮動小数点演算を用いて計算した $Az = \tilde{r}$ の数値解を $\tilde{z}$ とすると

$$x_{\mathrm{new}} = \tilde{x} \ominus \tilde{z} \tag{2.1.3}$$

によって，式 (1.1.1) の新しい数値解 $x_{\rm new}$ が得られる．これが本論文での残差反復法の実装法の提案である．

この方法では，多倍長浮動小数点数パッケージ等を利用せず，内積の高精度計算アルゴリズム **Dot2** のみを用いているため，IEEE754 の倍精度浮動小数点数規格に従う計算システムであればポータブルに実装できることがわかる．

定理 5 では $\tilde{x}$ は任意であるが，我々の実装法においては，2.1.1 節で提案した高精度内積計算アルゴリズム **Dot2** を用いた残差反復法により必要な回数だけ反復した式 (1.1.1) の近似解を取るものとする．また，式 (2.1.2) では，高精度な残差 $r = A\tilde{x} - b$ の近似値 $\tilde{r}$ を計算しているが，精度保証の過程では，さらに $|r - \tilde{r}| \leqq q$ を満たすようなベクトル q を求める必要がある．ただし，2 つのベクトル $x, y \in \mathbb{R}^n$ に対して $x \leqq y$ は，すべての i に対して $x_i \leqq y_i$ が成り立つことを意味する．

そこで，**DotEFT** 及び **SumEFT** を用いて高精度に残差の近似値とその誤差限界を求める MATLAB のアルゴリズムを実装例として次の節に示しておく．

2.2 精度保証の MATLAB プログラム

ここでは，実際の精度保証に用いたプログラムを示す[1]．これらは MATLAB の関数である．

以下は，それぞれ **TwoSum**，**Split** 及び **TwoProduct** である．

```
function [x,y] = TwoSum(a,b)
% TwoSum: error-free transformation of sum a + b to x + y.
  x = a + b;
  t = x - a;
  y = (a - (x - t)) + (b - t);

function [ah,al] = Split(a)
% Split: error-free transformation of a to ah + al.
  c = (2^27 + 1)*a;
  ah = c - (c - a);
  al = a - ah;

function [x,y] = TwoProduct(a,b)
% TwoProduct: error-free transformation of product a x b to x + y.
  x = a*b;
  [ah,al] = Split(a); [bh,bl] = Split(b);
  c1 = x - ah*bh; c2 = c1 - al*bh; c3 = c2 - ah*bl;
  y = al*bl - c3;
```

以下は，それぞれ **Dot2**，**SumEFT** 及び **DotEFT** である．

```
function res = Dot2(x,y)
%  Dot2: calculation of dot product as if computed in quadruple precision.
```

[1] %から行末まではコメント

```
  n = length(x);
  [p,s] = TwoProduct(x(1),y(1));
  for i=2:n
      [h,r] = TwoProduct(x(i),y(i));
      [p,q] = TwoSum(p,h);
      s = s + (q + r);
  end
  res = p + s;

function q = SumEFT(p)
%  SumEFT: error-free transformation of summation.
%    input
%      p: a real n-vector
%    output
%      q: a real n-vector s.t. sum(q) = sum(p)

  q = p;
  for i=2:length(q)
      [q(i),q(i-1)] = TwoSum(q(i),q(i-1));
  end

function v = DotEFT(x,y)
%  DotEFT: error-free transformation of dot product to summation.
%    input
%      x, y: real n-vectors
%    output
%      v: a real 2n-vector s.t. sum(v) = dot(x,y)

  n = length(x);
  v = zeros(2*n,1);
  [t,v(1)] = TwoProduct(x(1),y(1));
  for i=2:n
      [h,v(i)] = TwoProduct(x(i),y(i));
      [t,v(n+i-1)] = TwoSum(t,h);
  end
  v(2*n) = t;
```

残差反復法の MATLAB での実装法の例を以下に示す．ただし，アルゴリズム中の R 及び x_s は，それぞれ A の近似逆行列，$Ax = b$ の近似解を表す．

```
function xs = iter_ref(A,b,R,xs)
```

```
%  iter_ref: iterative refinement for an approximate solution of Ax = b
%            using accurate dot product.
%  input
%     A: a real n x n matrix, b: real n-vector
%     R: an approximate inverse of A, xs: an approximate solution of Ax = b
%  output
%     xs: an updated approximate solution of Ax = b

  [m,n] = size(A);  % size of A
  r = zeros(n,1);
  for i=1:n         % residual A*xs - b by Dot2
      r(i) = Dot2([A(i,:),b(i)]',[xs; -1]);
  end
  z = R*r;          % approximate solution of Az = r
  xs = xs - z;      % update xs
```

以下に，高精度な残差計算とその誤差限界を計算するプログラムを示す．ただし，

```
  system_dependent('setround',mode)
```

は丸めモードの変更をする命令で，mode = 'nearest' は最近点への丸めモードに，mode = -inf は，$-\infty$ 方向の丸めモードに，mode = +inf は $+\infty$ 方向の丸めモードにそれぞれ変更することを表す．

```
function [rmid,rrad] = accresidual(A,b,xs)
% accresidual: accurate computation of A*xs - b and its error bound.
%   input
%     A: a real n x n matrix, b: a real n-vector
%     xs: an approximate solution of Ax = b
%  output
%     rmid: an approximation of A*xs - b
%     rrad: an error bound of rmid

  system_dependent('setround','nearest');
  n = length(b);
  rmid = zeros(n,1);
  rrad = rmid;
  for i=1:n
      v = DotEFT([A(i,:),b(i)]',[xs; -1]);
      v = SumEFT(v);
      rmid(i) = v(end);
      system_dependent('setround',+inf);
      rrad(i) = sum(abs(v(1:end-1)));
```

```
    system_dependent('setround','nearest');
  end
```

最後に，連立一次方程式 $Ax = b$ の精度保証アルゴリズムを以下に示す.

```
function [e,alpha] = vlin(A,b,R,xs,mode,alpha)
%  vlin: fast verification of an approximate solution xs of Ax = b.
%  input
%     A: a real n x n matrix, b: real n-vector
%     R: an approximate inverse of A, xs: an approximate solution of Ax = b
%     mode: If mode = 0, then compute alpha s.t. alpha >= norm(RA - I,inf).
%           Otherwise, skip calculation of alpha.
%  output
%     e: an error bound of xs
%     alpha: alpha >= norm(RA - I,inf)

  [m,n] = size(A);
  if mode == 0
    I = speye(n);                       % I: n x n identity matrix
    system_dependent('setround',-inf);  % rounding to -infinity
    Gl = R*A - I;                       % lower bound of R*A - I
    system_dependent('setround',+inf);  % rounding to +infinity
    Gu = R*A - I;                       % upper bound of R*A - I
    Gu = max(abs(Gl),abs(Gu));          % upper bound of |R*A - I|
    alpha = norm(Gu,inf);               % alpha >= norm(RA - I,inf)
  end

  if alpha < 1
    [rrmid,rrad] = accresidual(A,b,xs); % accurate residual and its error bound
    system_dependent('setround',+inf);
    t = R*rrad;
    vl = R*rmid + t;
    system_dependent('setround',-inf);
    vu = R*rmid - t;
    vu = max(abs(vl),abs(vu));          % vu >= |R*(A*xs - b)|
    d = 1 - alpha;
    system_dependent('setround',+inf);
    e = norm(vu,inf)/d;                 % error bound of xs
    system_dependent('setround','nearest');
  else
    error('verification failed');
```

```
end
```

したがって，IEEE754 の倍精度浮動小数点数規格に従う計算システムであれば，本論文におけるアルゴリズムはすべてポータブルに実装できることがわかる．

2.3 数値例

ここでは，提案したアルゴリズムの有効性を確認するために行った数値実験の中から幾つかを紹介する．

2.3.1 悪条件の問題

行列の条件数が $\mathcal{O}(10^{14})$ 以上となる問題の例として，ヒルベルト行列を係数行列とする問題を考える．$n \times n$ ヒルベルト行列 H_n はその第 (i, j) 成分を h_{ij} とするとき

$$h_{ij} = \frac{1}{i + j - 1} \tag{2.3.1}$$

で定義される行列である．ここでは，$n = 11$ のヒルベルト行列 H_{11} に，要素 h_{ij} の分母の公倍数をかけた行列 K を係数行列とする連立一次方程式を考える．また，$x = (1, 1, \cdots, 1)^T$ が厳密解であるように右辺ベクトルは $b = fl(K \cdot (1, 1, \cdots, 1)^T)$ とする．このとき，丸め誤差は発生しない．すなわち，11 次元ベクトル x についての連立一次方程式

$$Kx = b \tag{2.3.2}$$

を考える．K の条件数は，$\mathrm{cond}_\infty(K) = \|K\|_\infty \|K^{-1}\|_\infty = 1.23 \cdots \times 10^{15}$ である．これを本論文で提案した高精度かつ高速な残差反復法と精度保証プログラムを用いて MATLAB 上で解いた例を以下に示す．

```
>> K = myhilb(11);     % K: the 11 x 11 Hilbert matrix
>> b = K*ones(11,1);   % b: a right-hand side vector
>> R = inv(K);         % R: an approximate inverse of K
>> xs = R*b;           % xs: an approximate solution of Kx = b
>> xs(11)
ans =
   0.99945068359375
>> [e,alpha] = vlin(K,b,R,xs,0)     % e: verified error bound of xs
e =
   0.01427499840825
alpha =
   0.04138183593750
>> xs = iter_ref(K,b,R,xs); xs(11)  % 1st iterative refinement of xs
ans =
   1.00000026822090
>> e = vlin(K,b,R,xs,1,alpha)
```

```
e =
    3.004930254062938e-006
>> xs = iter_ref(K,b,R,xs); xs(11)  % 2nd iterative refinement of xs
ans =
   1.00000000032014
>> e = vlin(K,b,R,xs,1,alpha)
e =
    1.220669425188106e-008
>> xs = iter_ref(K,b,R,xs); xs(11)  % 3rd iterative refinement of xs
ans =
   1.00000000000023
>> e = vlin(K,b,R,xs,1,alpha)
e =
    2.285908904331256e-011
>> xs = iter_ref(K,b,R,xs); xs(11)  % 4th iterative refinement of xs
ans =
   1.00000000000000
>> e = vlin(K,b,R,xs,1,alpha)
e =
    5.576489225820260e-014
>> xs = iter_ref(K,b,R,xs); xs(11)  % 5th iterative refinement of xs
ans =
     1
>> e = vlin(K,b,R,xs,1,alpha)
e =
    1.166207784303186e-016
>> xs = iter_ref(K,b,R,xs); xs(11)  % 6th iterative refinement of xs
ans =
     1
>> e = vlin(K,b,R,xs,1,alpha)
e =
     0
```

この例においては，`R = inv(K)` という命令によって，行列 K の逆行列を倍精度浮動小数点演算で求めている．丸め誤差のため，計算された R は K の近似逆行列になっている．式 (2.3.2) の初期近似解としては，方程式を倍精度浮動小数点演算で計算した解（`xs = R*b` という命令で求めた解）を採用している．尚，`xs(11)` という命令は，ベクトル x の第 11 番目の成分を表示せよとの命令になっている．これは，解がどのように収束するかを見るために，例として `xs` の一つの成分を表示させたものである．今の場合は，厳密解は $x = (1, 1, \cdots, 1)^T$ であるから，小数点以下 4 桁目以降に誤差があることが見て取れる．

次の命令では，計算した近似解 $\tilde{x} =$ xs のもつ精度を本論文で提案した方法に基づいて実装した高精度精度保証プログラム vlin によって計算している．その結果，誤差 $e = \|A^{-1}b - \tilde{x}\|_\infty$ は $1.427\ldots \times 10^{-2}$ 以下であるとの見積もりが得られている．

その次の命令では，2.1.1 節で提案した高精度内積計算法に基づいた残差反復法により，一回反復した xs を計算している．そして，その第 11 成分 xs(11) を表示している．この結果から，近似解の小数点以下 7 桁目に誤差があることが見て取れる．その次の命令ではその誤差を $\|A^{-1}b - x\|_\infty \leqq 3.004\ldots \times 10^{-6}$ 以下であるとの見積もりが得られている．

以下これが繰り返されている．一回の残差反復では 10^{-3} 程度のオーダで残差反復した解の正しい桁が増えていき，6 回目の反復で最後の桁（least significant digit）まで正しく計算されたことがわかる[2]．また，誤差評価もそのことを正しく評価できていることがわかる．

2.3.2 良条件の問題

条件数は $\mathcal{O}(10^{14})$ に比較して小さいが，問題の次元が $n = 1000$ と前節の問題に比して大きな問題を例として取り上げる．すなわち，A は 1000×1000 実行列でその要素は乱数であるとする．

```
>> n = 1000;
>> A = randn(n);      % A: an n x n random matrix
>> b = A*ones(n,1);
>> cond(A)            % condition number estimator
ans =
    2.450763206290818e+004
```

より，A の条件数は $\mathcal{O}(10^4)$ 程度である．b = A*ones(n,1); からわかるように，$b = A\cdot(1,1,\cdots,1)^T$ として式 (1.1.1) を考えている．ただし，丸め誤差の存在により，この場合の式 (1.1.1) の厳密解は $x = (1,1,\cdots,1)^T$ と若干異なることに注意する．

このようにして作成した A と b を用いて，$Ax = b$ の精度保証を実行する．

```
>> R = inv(A);
>> xs = R*b; xs(1000)
ans =
   1.00000000000102
>> [e,alpha] = vlin(A,b,R,xs,0)
e =
    2.219658324896038e-011
alpha =
    8.249229812288443e-010
>> xs = iter_ref(A,b,R,xs); xs(1000)
ans =
   1.00000000000002
```

[2]誤差限界 $e = 0$ ということは，近似解 $\tilde{x}$ は厳密解そのものであることを意味する．

表 2.1: 精度保証のための計算時間 [sec]（第 3,4 列における括弧内の数字は LU 分解の時間に対して何倍計算時間を要しているかを示している）

n	LU 分解	標準精度保証法	新精度保証法
200	0.02	0.04 (2.00)	0.04 (2.00)
300	0.04	0.14 (3.50)	0.17 (4.25)
400	0.07	0.31 (4.42)	0.35 (5.00)
500	0.13	0.50 (3.85)	0.63 (4.84)
600	0.21	0.87 (4.14)	1.01 (4.81)
700	0.30	1.31 (4.37)	1.53 (5.10)
800	0.45	2.23 (4.95)	2.37 (5.23)
900	0.59	2.89 (4.90)	3.07 (5.20)
1000	0.81	3.84 (4.74)	4.31 (5.32)

```
>> e = vlin(A,b,R,xs,1,alpha)
e =
    1.107097103095627e-016
```

この例では，一回の残差反復でほぼ最終桁まで正しい結果が得られていることがわかる．また，本論文で提案している高精度精度保証プログラムがシャープな誤差評価を与えていることがわかる．この点に関し，残差を高精度に計算しない従来の精度保証プログラム `vlin_pre` で精度を計算すると

```
>> e = vlin_pre(A,b,R,xs)
e =
    4.566030510093495e-011
```

となって，本来，近似解が持っている精度をシャープに評価できていないことがわかる．

ここで，式 (1.1.1) の近似解を求める時間 (A の LU 分解にかかる時間) と残差を高精度に計算しない従来の精度保証プログラムで精度保証する時間 (本論文ではこれを標準精度保証法の計算時間という) と残差を高精度に計算する本論文で提案した方法での時間 (新法の計算時間と呼ぶ) を表 2.1 に示す．表中に示した数字は各次元とも，異なった 100 のランダム行列 A を生成し，$b = A \cdot (1, 1, \cdots, 1)^T$ として，式 (1.1.1) を解いた時間を平均したものである．尚，精度保証の時間には A の近似逆行列 R を計算する時間を含めていない．計算に用いたコンピュータは Mac Power Book G4 (1.25GHz CPU) で計算に用いた言語は Slab [4] である．Slab は MATLAB に似た数値計算ツールだが，精度保証ができるように改良されたものであり，MATLAB と同様，LAPACK をもとにして，これを使いやすくするインタプリタとなっている．

近似解 (LU 分解) を計算する計算のため浮動小数点演算の回数 (flops という単位が標準で用いられる．floating point operations の略である．) は四則演算をすべて一回と数えると $2/3n^3$ flops である．一方，標準精度保証法も新精度保証法も近似逆行列 R を計算する時間は含めていないので，行列の積を 2 回計算する手間の $4n^3$ flops である．よって，浮動小数点演算の回数の比でいえば，精度保証にかかる計算時間は標準法も新しい方法もともに LU 分解法の計算時間の 6 倍となる．表 2.1 に示した実際の数値計算例において

はこの比は 2 つの精度保証法とも 6 倍より少なくなっている．これは行列の積の計算が ATLAS によって生成された最適化 BLAS を基に行われていることに起因する．すなわち，最適化 BLAS はキャッシュヒット率を最大にするように行列の積を計算するように生成されている．ガウスの消去法の中にもこの最適化は生かされているが，行列の積の方が単純なアルゴリズムで計算できるので，この最適化の効果がより生かされる．したがって，精度保証に必要な主計算が行列の積であることから，実際の計算時間が浮動小数点演算の回数の比よりは短縮されているのである．

また，標準精度保証法の計算時間と新しい精度保証法の計算時間は，後者が高精度な内積計算を残差の計算に用いているので，よりかかっているが，ほぼ同じオーダーであることが確認された．すなわち，残差の計算において高精度な内積計算を用いてもその演算回数は $\mathcal{O}(n^2)$ であり，しかも高精度な内積計算法に要する時間が浮動小数点演算での内積計算に要する時間の一定の倍率でしか時間がかからないことからこれは理論的にも理解される．すなわち，新しい精度保証法は標準精度保証法の高速性を損なうことなく，高精度性が達成されていることがわかる．

第3章　条件数が非常に大きな係数行列をもつ連立一次方程式の精度保証付き数値計算法

本章では，任意に条件数の大きい係数行列を持つ連立一次方程式の数値解の精度保証法について論じる．提案する方法は，高精度内積計算法と IEEE754 規格に基づいている．

式 (1.1.1) に対し

$$\kappa(A) := \|A\| \cdot \|A^{-1}\|$$

をその条件数という．式 (1.1.1) の厳密解を $x^* := A^{-1}b$ とすると (ただし，A が正則のとき)，b が $b + \Delta b$ に変化したとき，解は $\kappa(A)\|\Delta b\|$ のオーダで変化する．すなわち，解の変化量を Δx として連立一次方程式 $A(x + \Delta x) = b + \Delta b$ を考えると

$$\frac{\|\Delta x\|}{\|x^*\|} \leq \kappa(A)\frac{\|\Delta b\|}{\|b\|} \tag{3.0.1}$$

が成り立つ．したがって，条件数の大きな問題では，右辺ベクトルの少しの変化が解の大きな変化を引き起こすことが分かる (条件数についてのより多くの考察については文献 [3] を参照されたい)．IEEE754 規格の倍精度浮動小数点数演算で計算する場合，仮数部の相対精度は高々$2^{-53} = 1.11\cdots\times 10^{-16}$ であるから，条件数が 10^{16} 程度の大きさになると，丸め誤差の混入する数値計算での数値解の精度は仮数部で 1 桁あるかないかというオーダになり，倍精度演算の限界となる．ここでは，10^{16} 程度より大きな条件数を持つ問題を悪条件な問題と呼んでいる．本論文では，条件数の非常に大きい行列に対して，その逆行列を高精度に求める方法やそれを応用した連立一次方程式の数値解の精度保証法について検討する．倍精度演算の範囲で取り扱うことができない問題では高精度演算が不可欠であるが，すべての計算に対して多倍長演算などを適用するのは計算コスト的に有効ではない．一方で，内積計算や行列積など特定の計算に限れば，それらの高精度演算について比較的高速なアルゴリズムが利用可能である (たとえば，文献 [6, 7, 9, 11])．そこで，本論文では，与えられた行列の条件数が非常に大きい場合でも，内積計算 (行列・ベクトル積や行列積を含む) さえ高精度に実行できれば，その他には多倍長演算などを必要としない高速な精度保証法を提案する．また，数値例によりその有効性を示す．

3.1　高精度な内積計算

本論文では，連立一次方程式 $Ax = b$ の係数行列 A と右辺ベクトル b の各要素は IEEE754 規格に従う浮動小数点数とし，解の計算には浮動小数点数演算を用いるという立場を取る．

本論文で提案する連立一次方程式のための精度保証法は内積計算のアルゴリズムに依存しない．ここでは，完全精度内積演算のように正確に求めた内積の値を必要な精度だけ得ることができるという仮定のもとに議論を進める．

汎用的な関数として

$$\mathbf{DotExact}(expression, parameter)$$

は，条件 *parameter* に従って *expression* を高精度に計算するものとする．たとえば，$A \in \mathbb{F}^{m\times p}, B \in \mathbb{F}^{p\times n}$ に対して行列乗算 $A \cdot B$ を考えたとき

$$C_{1:k} := C_1 + C_2 + \cdots + C_k = \mathbf{DotExact}(A \cdot B, k)$$

は

$$|C_i| \geq 2^{52} \cdot |C_{i+1}|, \quad i = 1, 2, \ldots, k-1$$

であり (つまり，仮数部がほとんど影響し合わない順番に並んでいる)，かつ

$$|\sum_{i=1}^{k} C_i - A \cdot B| \leq \max(2^{-52} \cdot |C_k|, realmin \cdot E) \tag{3.1.1}$$

となるような $C_i \in \mathbb{F}^{m\times n}, (i = 1, 2, \ldots, k)$ を求めるものとする．ただし，$realmin := 2^{-1022}$ は IEEE754 倍精度浮動小数点数で表現できる正規化数のうち正の最小のものであり，E はすべての要素が 1 の $m \times n$ 行列とする．これらは，アンダフローが起きたときにも精度保証が厳密であるために必要となる．また

$$[C_{1:k}, C_{\mathrm{rad}}] = \mathbf{DotExact}(A \cdot B, \{\text{'midrad'}, k\})$$

は

$$|C_i| \geq 2^{52} \cdot |C_{i+1}|, \quad i = 1, 2, \ldots, k-1$$

かつ

$$|\sum_{i=1}^{k} C_i - A \cdot B| \leq C_{\mathrm{rad}}$$

となるような $C_i \in \mathbb{F}^{m\times n}, (i = 1, 2, \ldots, k)$ 及び $C_{\mathrm{rad}} \in \mathbb{F}^{m\times n}$ を求めるものとする．さらに

$$A_{1:k} := A_1 + A_2 + \ldots + A_k = \sum_{i=1}^{k} A_i, \quad A_i \in \mathbb{F}^{n\times n}$$

に対しても

$$\mathbf{DotExact}(A_{1:k} \cdot B, parameter)$$

のような使い方ができるものとする．これは

$$\mathbf{DotExact}(A_1B + A_2B + \ldots + A_kB, parameter)$$

と同値である．同様に

$$\mathbf{DotExact}(A_{1:k_1} \cdot B_{1:k_2}, parameter)$$

は

$$\mathbf{DotExact}(A_1B_1 + A_2B_1 + \ldots + A_{k_1}B_{k_2}, parameter)$$

を意味するものとする．

3.2 提案方式

3.2.1 成分ごとの精度保証法

近年，Oishi と Rump [10] は式 (1.1.1) の数値解に対する高速精度保証法を提案した．数値解に対して成分ごとに誤差評価を行うには次の Yamamoto の定理 [14] が有用である．解ベクトルの要素において，その絶対値の大きさにばらつきがあるときは，成分ごとの誤差評価を行うことにより絶対値が相対的に小さい要素に対しても過大評価の少ない精度保証が可能となる．

定理 6 (Yamamoto [14]) A を $n \times n$ 実行列，b を n 次元実ベクトルとして連立一次方程式

$$Ax = b$$

を考える．適当な $n \times n$ 実行列 R が存在して[1]

$$\|RA - I\|_\infty < 1 \tag{3.2.1}$$

を満たすとき，A は正則で，任意の n 次元実ベクトル $\tilde{x}$ に対して[2]

$$|\tilde{x} - A^{-1}b| \leq |R(A\tilde{x} - b)| + \frac{\|R(A\tilde{x} - b)\|_\infty}{1 - \|RA - I\|_\infty} t \tag{3.2.2}$$

が成り立つ．ただし，t は $G := RA - I$ としたときに

$$\left(\sum_{j=1}^{n} |G_{1j}|, \sum_{j=1}^{n} |G_{2j}|, \ldots, \sum_{j=1}^{n} |G_{nj}|\right)^T \leq t \tag{3.2.3}$$

を満たす n 次元ベクトルであり，I は n 次元単位行列である． □

この定理では $\tilde{x}$ は任意であるが，我々の実装法においては，3.1 節で紹介した高精度内積計算アルゴリズム **DotExact** を用いた残差反復法により，必要な回数だけ反復した式 (1.1.1) の近似解を取るものとする．

具体的には，式 (3.2.2) の右辺の分子に現れる $A\tilde{x} - b$ を **DotExact** を用いて高精度に計算する．R についても，**DotExact** を用いた反復計算により $\|RA - I\|_\infty < 1$ を満たすような R を求める．

提案する精度保証法の詳細については，次の節で述べる．

3.2.2 条件数の大きい行列に対する精度保証法

A の条件数が 10^{16} 程度より大きいと，倍精度演算の範囲では $\|RA - I\|_\infty$ の値が 1 を超えてしまい，3.2.1 節で示した精度保証法が適用できなくなる．これは，計算中の丸め誤差の影響や，R の表現に必要な精度の不足によって，R が A の逆行列としての情報をほとんど持たないことを意味する．しかしながら，そのような場合においても，実は R は A の前処理としての情報を依然として含んでいることが知られている [12]．この性質を利用して

$$\|RA - I\|_\infty < 1$$

[1] 実際の応用では R としては A の近似逆行列を取るのが普通である．
[2] やはり実際の応用では $Ax = b$ の近似解を $\tilde{x}$ とする．

を満たすような R を高精度に求める方法を提案する．

まず，倍精度演算の性質をできるだけ有効に活用するために，R を

$$R = R_{1:k} = R_1 + R_2 + \ldots + R_k = \sum_{i=1}^{k} R_i, \quad R_i \in \mathbb{F}^{n\times n}$$

のように，要素が倍精度浮動小数点数である行列の和として表現することを考える．ただし

$$|R_i| \geq 2^{52} \cdot |R_{i+1}|, \quad i = 1, 2, \ldots, k-1$$

とする．ここで，我々は文献 [12] の方法を拡張した次のアルゴリズムを提案する．尚，本論文ではアルゴリズムを MATLAB に近い表記で記述する．

アルゴリズム 4 高精度に $R = \sum_{i=1}^{k} R_i$ を求めるアルゴリズム：

$R_1^{(1)} = \mathbf{inv}(A)$ % A の逆行列の計算 (倍精度演算)
for $i = 2 : k$
  $C = \mathbf{DotExact}(R_{1:i-1}^{(i-1)} \cdot A, 1)$ % $C \leftarrow R \cdot A$ (高精度演算, 結果は倍精度)
  $T = \mathbf{inv}(C)$ % $T \approx C^{-1}$ (倍精度演算)
  $R_{1:i}^{(i)} = \mathbf{DotExact}(T \cdot R_{1:i-1}^{(i-1)}, i)$ % $R_{new} \leftarrow T \cdot R$ (高精度演算)
end

まず，$R_1^{(1)}$ を A の最初の前処理行列と考え，$R_1^{(1)}A$ を高精度に計算し，結果を倍精度に丸めて

$$C \approx R_1^{(1)} A$$

のようにする．次に，C の (近似) 逆行列を通常の倍精度演算で

$$T \approx C^{-1} \approx (R_1^{(1)} A)^{-1}$$

のように求める．そして，$TR_1^{(1)}$ を高精度に計算し，新たに求めた $R_1^{(2)}, R_2^{(2)}$ を次の前処理行列 $R_{1:2}^{(2)} := R_1^{(2)} + R_2^{(2)}$ とすると，結果は

$$R_{1:2}^{(2)} = R_1^{(2)} + R_2^{(2)} \approx TR_1^{(1)} \approx (R_1^{(1)} A)^{-1} R_1^{(1)} = A^{-1}$$

となることが期待できる．すなわち，C の条件数が A の条件数よりも小さくなり，T と $(R_1^{(1)}A)^{-1}$ との差は $R_1^{(1)}$ と A^{-1} との差よりもかなり小さくなると期待されるので，A の悪条件性が改善される．同様に上記の操作を繰り返し，$R_{1:k}^{(k)} := R_1^{(k)} + R_2^{(k)} + \ldots + R_k^{(k)}$ を求める．このアルゴリズムにより，通常の倍精度演算と比較して高精度な R を計算することが可能となり，実際，条件数 $\kappa(A)$ が

$$\kappa(A) \lesssim (c_n^{-1} \cdot 10^{16})^k \tag{3.2.4}$$

程度の問題であれば A の正則性の保証が可能となる (これについては，後で数値実験によって確認する)．ただし，一般的には A の条件数は未知であるため，計算された R が $\|RA - I\|_\infty < 1$ を満たすかどうかは保証されない．すなわち，事前に R にはどれだけの精度が必要か (k をいくつにすれば良いか) はわからないため，A の性質によって $\|RA - I\|_\infty < 1$ を満たすような R を自動的に反復改良しながら計算できるアルゴリズムが望ましい．

そこで，反復ごとに $\|RA - I\|_\infty < 1$ の判定をしながら R を更新していくアルゴリズムを以下に示す．また，定理 6 を適用するため式 (3.2.3) を満たす t を求める操作も加える．

アルゴリズム 5 $\|RA-I\|_\infty \le \alpha < 1$ を満たす R, α 及び式 (3.2.3) を満たす t を求めるアルゴリズム：

function $[R_{1:k}^{(k)}, \alpha, t] = \mathbf{AccInv}(A, k_{\max})$
$k = 1$
$R_1^{(1)} = \mathbf{inv}(A)$
$\underline{G} = fl_\triangledown(R_1^{(1)}A - I)$ % $RA-I$ の下限
$\overline{G} = fl_\triangle(R_1^{(1)}A - I)$ % $RA-I$ の上限
$\overline{G} = \max(|\underline{G}|, |\overline{G}|)$ % $|RA-I|$ の上限
for $i = 1 : n$
  $t_i = fl_\triangle\left(\sum_{j=1}^n \overline{G}_{ij}\right)$ % $\sum_{j=1}^{n} \overline{G}_{ij} \le t_i$
end
$\alpha = \max_{1\le i\le n} t_i$ % $\|RA-I\|_\infty$ の上限
if $\alpha < 1,$ return, end
while $k < k_{\max}$ % $k_{\max}$: 最大反復回数
  $C = \mathbf{DotExact}(R_{1:k}^{(k)} \cdot A, 1)$
  if $fl_\square(\|C-I\|_\infty) < 10^{-3},$ break, end % 反復停止条件
  $T = \mathbf{inv}(C)$
  $R_{1:k+1}^{(k+1)} = \mathbf{DotExact}(T \cdot R_{1:k}^{(k)}, k+1)$
  $k = k + 1$
end
$S = fl_\square(\max(2^{-52} \cdot |C|, realmin \cdot E))$ % $S_{ij} = \max(2^{-52} \cdot |C_{ij}|, realmin)$
$\underline{G} = fl_\triangledown(C - I - S)$ % $RA-I$ の下限
$\overline{G} = fl_\triangle(C - I + S)$ % $RA-I$ の上限
$\overline{G} = \max(|\underline{G}|, |\overline{G}|)$ % $|RA-I|$ の上限
for $i = 1 : n$
  $t_i = fl_\triangle\left(\sum_{j=1}^n \overline{G}_{ij}\right)$ % $\sum_{j=1}^{n} \overline{G}_{ij} \le t_i$
end
$\alpha = \max_{1\le i\le n} t_i$ % $\|RA-I\|_\infty$ の上限
if $\alpha \ge 1,$ error('verification failed.'), end

式 (3.1.1) から

$$C = \mathbf{DotExact}(R_{1:k}^{(k)} \cdot A, 1)$$

によって得られる C は

$$|C - R_{1:k}^{(k)} \cdot A| \le \max(2^{-52} \cdot |C|, realmin \cdot E) =: S$$

を満たすため，$R := R_{1:k}^{(k)}$ とすると

$$C - I - S \le RA - I \le C - I + S$$

が成り立つ．よって，$\|RA-I\|_\infty$ の上限 α を計算可能となる．最後に，α が 1 より小さいかどうかを確認すれば良い．

これにより，反復の停止条件を考慮しても，1 反復あたりの高精度な行列乗算 (**DotExact**) の回数は 2 回で済むことが分かる．そして，アルゴリズム 5 によって A の正則性が保証されると，連立一次方程式 $Ax = b$ の精度保証が可能となる．

3.2.3 連立一次方程式の精度保証

連立一次方程式 $Ax = b$ の近似解 $\tilde{x}$ に対する残差を $r := A\tilde{x} - b$ とすると，厳密解 $x^* := A^{-1}b$ に対する $\tilde{x}$ の誤差は，連立一次方程式 $Ap = r$ の解 $p^* := A^{-1}r$ で与えられることから ($x^* = \tilde{x} - p^*$)，残差反復は以下のような手順で実行できる．ただし，A が正則のとき，r の要素がすべてゼロである場合は $\tilde{x}$ が $Ax = b$ の厳密解であるため，$\|r\| \neq 0$ と仮定する．

1. 残差 $A\tilde{x} - b$ を高精度に計算する ($\tilde{r} \leftarrow A\tilde{x} - b$).
2. 連立一次方程式 $Ap = \tilde{r}$ を解く (R を用いて $\tilde{p} \leftarrow R \cdot \tilde{r}$ とする).
3. 近似解 $\tilde{x}$ を $\tilde{x}_{new} \leftarrow \tilde{x} - \tilde{p}$ と更新する．

我々は，この残差反復法と精度保証を組み合わせることにより，所望の相対精度 tol を与えたときに

$$\left| \frac{\tilde{x}_i - x_i^*}{\tilde{x}_i} \right| \leq \text{tol}, \quad (\tilde{x}_i \neq 0 \text{ のとき}) \tag{3.2.5}$$

が成り立つような $\tilde{x} \in \mathbb{F}^n$ と

$$|\tilde{x} - x^*| \leq y \tag{3.2.6}$$

を満たす $y \in \mathbb{F}^n$ を求める方法を提案する．

このときに注意することは，$Ap = r$ の右辺に Δr だけ摂動を加えた連立一次方程式 $A(p + \Delta p) = r + \Delta r$ を考えると，式 (3.0.1) と同様に

$$\frac{\|\Delta p\|}{\|p^*\|} \leq \kappa(A) \frac{\|\Delta r\|}{\|r\|}$$

が成り立つことである．つまり，A の条件数を考慮して

$$\kappa(A)\|\Delta r\| \ll \|r\|$$

を満たすように，残差 $A\tilde{x} - b$ を高精度に計算し，その結果を高精度のまま保持する必要がある．

以上の議論を考慮して，Yamamoto の定理に基づく連立一次方程式の数値解の精度保証アルゴリズムを以下に示す．

アルゴリズム 6 $A \in \mathbb{F}^{n \times n}$ の正則性を判定し，正則であることが保証できたときに，式 (3.2.5) を満たす $Ax = b$ の近似解 $\tilde{x}$ と $|\tilde{x} - A^{-1}b|$ の上限 $y \in \mathbb{F}^n$ を求めるアルゴリズム：

function $[\tilde{x}, y] = \mathbf{AccVerLin}(A, b, \mathrm{tol}, k_{\max})$
$[R_{1:k}, \alpha, t] = \mathbf{AccInv}(A, k_{\max})$ % $\|RA - I\|_\infty \le \alpha$
$\tilde{x} = \mathbf{DotExact}(R \cdot b, 1)$ % $\tilde{x} \leftarrow R \cdot b$ (高精度，結果は倍精度)
for loop $= 1 : 10$ % 残差反復
  $[r^{(1:k)}, r_{\mathrm{rad}}] = \mathbf{DotExact}(A\tilde{x} - b, \{\text{'midrad'}, k\})$ % $|r^{(1:k)} - (A\tilde{x} - b)| \le r_{\mathrm{rad}}$ (高精度)
  $[p, p_{\mathrm{rad}}] = \mathbf{DotExact}(R_{1:k} \cdot r^{(1:k)}, \{\text{'midrad'}, 1\})$ % $|p - R_{1:k} \cdot r^{(1:k)}| \le p_{\mathrm{rad}}$ (高精度)
  $q = fl_\triangle(|p| + (p_{\mathrm{rad}} + (\sum_{i=1}^{k} |R_i|) \cdot r_{\mathrm{rad}}))$
  $y = fl_\triangle(q + (\|q\|_\infty / -(\alpha - 1)) \cdot t)$ % 式 (3.2.2)
  $\mathrm{relerr} = fl_\triangle(\max_{1 \le i \le n, x_i \ne 0} |y_i / x_i|)$ % 相対誤差の最大値
  if relerr $\le$ tol % 残差反復の停止条件
    break
  else
    $\tilde{x} = \tilde{x} - p$ % 近似解 $\tilde{x}$ の更新
  end
end

ここで，我々の提案するアルゴリズムの特徴を以下にまとめる．

- 事前に R の計算に必要な精度がわからなくても，反復計算により問題に応じて精度を増やしながら必要なだけの計算をする
- 高精度演算が必要なのは基本的に行列積と行列・ベクトル積のみ
- したがって高速に実装可能
- ベクトル b を変更しても行列 A が同じならば逆行列 R と $\|RA - I\|_\infty$ の計算はやり直さなくても済む。

次節では，数値実験によって提案方式の有効性を検証する．

3.3 数値実験

本節では条件数の大きい問題に対して提案方式を適用し，その性能を評価する．数値実験に使用した計算機の CPU は Pentium 4 2.53GHz である．計算はすべて MATLAB (7.0.1.24704 (R14) Service Pack 1) 上で実行した．

3.3.1 ヒルベルト行列

まず，条件数が 10^{16} 以上になる問題の例として，ヒルベルト行列を考える．$n \times n$ ヒルベルト行列 $H = (h_{ij})$ は，その第 (i, j) 成分を h_{ij} とするとき

$$h_{ij} = \frac{1}{i + j - 1}, \quad (1 \le i, j \le n)$$

で定義される行列である．ここでは，係数行列に誤差が含まれないように，ヒルベルト行列 H に 1 から $2n-1$ までの公倍数 s をかけて行列 $A := s \cdot H$ を作る[3]．右辺ベクトルについては，$z \in \mathbb{F}^n$ を $z_i = (-1)^i$ として，$b := fl_{\square}(A \cdot z)$ と決める．このとき，b の計算に誤差は生じないため (すなわち，$fl_{\square}(A \cdot z) = Az$)，$Ax = b$ の厳密解は $x^* = z$ となる．

以上の問題に対し，$n = 20$ として，本論文で提案する精度保証法を用いて解いた例を以下に示す．このとき，A の条件数は $\kappa(A) = 2.45 \cdots \times 10^{28}$ となる．ただし，アルゴリズム 6 において $\mathrm{tol} = 10^{-9}$，$k_{\max} = 20$ とした．

```
>> n = 20; [A,b,cnd] = myhilb(n); cnd
cnd =
    2.452156585815303e+028
>> [x,y] = AccVerLin(A,b,1e-9,20); [x,y]
*** calculation of approximate inverse ***
k =   2
*** verification for a linear system with iterative refinement ***
loop =   1, max. rel. error = 1.095537e-004
loop =   2, max. rel. error = 8.874301e-010
ans =
   -1.000000000000000e+000    8.617077495942631e-025
    1.000000000000000e+000    9.640416868093147e-024
   -1.000000000000000e+000    1.062220917857571e-021
    1.000000000000000e+000    4.941914472774893e-020
   -9.999999999999992e-001    7.780126726211439e-016
    9.999999999999695e-001    3.054625398880036e-014
   -9.999999999994678e-001    5.324606289090067e-013
    9.999999999996297e-001    3.709046559563383e-013
   -9.999999999960786e-001    3.925436285241176e-012
    9.999999999602059e-001    3.980824821272961e-011
   -1.000000000120782e+000    1.208330840697098e-010
    1.000000000595174e+000    5.953338716910959e-010
   -9.999999998351981e-001    1.649529844243300e-010
    9.999999994154054e-001    5.849743981363064e-010
   -1.000000000637741e+000    6.379808828181332e-010
    9.999999991127588e-001    8.874300871906823e-010
   -9.999999999849613e-001    1.513451257318180e-011
    9.999999999542837e-001    4.574802156996649e-011
   -9.999999999890732e-001    1.093710468307201e-011
    9.999999999997991e-001    2.018749118516648e-013
```

ここで，ans の第 1 列は数値解 $\tilde{x}$，第 2 列は式 (3.2.6) の右辺，つまり $\tilde{x}$ の誤差の上限 y を示している．す

[3] n が大きくなると，この方法でも A は誤差を持つが，少なくとも $n \leq 20$ のときは A に誤差は含まれない．

なわち，$\tilde{x} - y \leq x^* \leq \tilde{x} + y$ である．`loop = 1` の時は残差反復をしていない場合の数値解，`loop = 2` 以降は (`loop` -1) 回残差反復をした場合の数値解を示している．`max. rel. error` は，$\tilde{x}$ の最大相対誤差 $\max_{1 \leq i \leq n} |\frac{\tilde{x}_i - x_i^*}{\tilde{x}_i}|$ の上限である．

以上の結果から，近似解 $\tilde{x}$ は残差反復により所望の精度を持つまで改善され，さらに精度保証付きで計算されていることが分かる．

次に，右辺ベクトルを $b := (1, 1, \ldots, 1)^T \in \mathbb{F}^n$ と変更した場合の結果を以下に示す．ここでは，$\mathrm{tol} = 10^{-12}$，$k_{\max} = 20$ とした．

```
>> b = ones(n,1);
>> [x,y] = AccVerLin(A,b,1e-12,20); [x,y]
*** calculation of approximate inverse ***
k =   2
*** verification for a linear system with iterative refinement ***
loop =   1, max. rel. error = 9.044724e-004
loop =   2, max. rel. error = 3.366434e-009
loop =   3, max. rel. error = 1.366729e-014
ans =
   -3.743263442685729e-015    5.116025474522475e-029
    1.493562113631585e-012    2.810456890124704e-028
   -1.478626492495270e-010    9.042732864902390e-027
    6.423810650729449e-009    8.237090906738374e-025
   -1.541714556175068e-007    2.638484718503020e-023
    2.312571834262602e-006    1.840219489496836e-022
   -2.338267076865519e-005    2.660917694689973e-021
    1.674962742815913e-004    2.209978255624679e-020
   -8.793554399783543e-004    5.504061868858120e-020
    3.463140559914753e-003    4.408457070587217e-020
   -1.038942167974426e-002    1.322862635541068e-019
    2.395577395577395e-002    3.086455593798719e-018
   -4.258804258804259e-002    1.631599120996661e-018
    5.821205821205821e-002    3.709033103548610e-018
   -6.058806058806059e-002    2.585259339885572e-018
    4.712404712404712e-002    2.010745751973094e-018
   -2.650727650727651e-002    1.998396005246811e-018
    1.018099547511312e-002    2.591511913013159e-019
   -2.388134741075917e-003    2.444648412569506e-019
    2.579979360165119e-004    2.070310897019167e-020
```

以上の結果から，近似解 $\tilde{x}$ の要素の大きさにばらつきがあったとしても，過大評価の少ない精度保証が要素ごとにできていることが分かる．

3.3.2 より条件数の大きい行列

次に，条件数がさらに大きい問題の例として，文献 [13] にある条件数を任意に設定できる手法を用いて得られた行列を係数行列 $A \in \mathbb{F}^{n\times n}$ とする．右辺ベクトルについては，$b := (1, 1, \ldots, 1)^T \in \mathbb{F}^n$ と決める．

$n = 100$，$\kappa(A) \approx 10^{100}$ のとき，提案する精度保証法による結果を以下に示す．

ただし，$\mathrm{tol} = 10^{-12}$，$k_{\max} = 20$ とした．ここで，式 (3.2.4) を考えると，$100/16 = 6.25$ から，反復回数 k は 7〜8 回くらいになることが予測される．

```
>> n = 100; A = randmat(n,1e+100); b = ones(n,1);
>> tic; [x,y] = AccVerLin(A,b,1e-12,20); toc
*** calculation of approximate inverse ***
k =   8
*** verification for a linear system with iterative refinement ***
loop =   1, max. rel. error = 6.921332e-006
loop =   2, max. rel. error = 4.789042e-011
loop =   3, max. rel. error = 4.264335e-016
経過時間は 8.109000 秒です
```

ここで，比較のために多倍長精度演算を用いた場合の結果を例として示す．MATLAB の toolbox である Symbolic Math Toolbox を用いると，VPA (Variable-Precision Arithmetic) という Maple[4]の任意多倍長演算ライブラリが MATLAB 上で利用可能となる．ここで，このような多倍長演算による計算の場合

- $\|RA - I\|_\infty \geq 1$ であった場合に R の精度を上げるためには，計算精度を増やして R の計算を最初からやりなおす必要がある
- $\|RA - I\|_\infty$ の計算自体にも膨大な計算時間が必要 (精度保証にはさらに計算時間が必要)

といった不利な点があることに注意する．今回は，VPA を用いて計算精度を 10 進で 32 桁，64 桁，128 桁と変更して R を計算し，さらにそれぞれ求まった R に対して高精度に $\beta \approx \|RA - I\|_\infty$ を計算したときの結果を示す．また，提案方式 (アルゴリズム 5) による結果も示す．

```
>> digits(32)                          % 計算精度: 10 進 32 桁
>> tic; R = inv(vpa(A)); toc           % A を多倍長精度にして，逆行列を計算
経過時間は 47.485798 秒です
>> tic; beta = norm(double(R*A-eye(n)),inf), toc
beta =
  5.3092e+004
経過時間は 98.243916 秒です

>> digits(64)                          % 計算精度: 10 進 64 桁
>> tic; R=inv(vpa(A)); toc
```

[4]数式処理システムのひとつ．

```
経過時間は 50.240413 秒です
>> tic; beta = norm(double(R*A-eye(n)),inf), toc
beta =
  3.2802e+003
経過時間は 147.076183 秒です

>> digits(128)                          % 計算精度：10 進 128 桁
>> tic; R=inv(vpa(A)); toc
経過時間は 73.390622 秒です
>> tic; beta = norm(double(R*A-eye(n)),inf), toc
beta =
  3.0993e-020
経過時間は 285.112813 秒です

>> tic; [R,alpha] = AccInv(A); toc  % 提案方式（アルゴリズム 2）
k =   8
経過時間は 6.642486 秒です
alpha =
  1.1690e-008
```

したがって，R の計算時間だけを考えても，単純に多倍長精度演算を用いるのに比べて，提案方式を用いると 10 倍程度は高速に $\|RA - I\|_\infty < 1$ を満たすような R を (保証付きで) 計算できていることが分かる．また，多倍長演算による $\beta \approx \|RA - I\|_\infty$ の (近似) 計算に必要な計算時間について考えると，本提案方式の有用性は明白であろう．ただし，オーバフローが起きるような，数値が倍精度の範囲を越えるような問題などでは，多倍長演算が必要になる場合もある．

最後に，問題サイズをもう少し大きくして，$n = 500$，$\kappa(A) \approx 10^{50}$ としたときに，提案する精度保証法を用いたときの結果を以下に示す．ただし，$\mathrm{tol} = 10^{-12}$，$k_{\max} = 20$ とした．ここで，式 (3.2.4) を考えると，$50/16 = 3.125$ であるから，反復回数 k は 3〜5 回くらいになることが予測される．

```
>> n = 500; A = randmat(n,1e+50); b = ones(n,1);
>> tic; [x,y] = AccVerLin(A,b,1e-12,20); toc
*** calculation of approximate inverse ***
k =   5
*** verification for a linear system with iterative refinement ***
loop =   1, max. rel. error = 3.776758e-009
loop =   2, max. rel. error = 1.023496e-016
経過時間は 190.500000 秒です
```

以上の結果から，A の条件数が非常に大きくなった場合や問題サイズ n が大きくなった場合でも，A の正則性の検証と連立一次方程式 $Ax = b$ の残差反復による近似解 $\tilde{x}$ の計算及びその精度保証を高速に実行可能であることが確認できた．

3.4 誤差評価をシャープに行うための工夫

解が大小ばらばらになった時には成分ごと誤差評価を行う事により絶対値の小さい解に対する精度保証がよりシャープに行うことができる．しかし，絶対値が最大の成分が影響して絶対値の小さな成分がシャープに評価出来なくなった．そこで，$Ax = b$ の代わりに，解 x の成分の絶対値がほぼ同じになる様に 2 の整数乗の z_i を定め

$$\Big(z_i a_{ij}\Big)x = b$$

の成分ごと誤差評価を行い，$|\tilde{x} - \Big(z_i a_{ij}\Big)^{-1} b| \leq p$ を求め

$$A(z_i x_i) = \Big(z_i a_{ij}\Big)x = b$$

により $|(z_i\tilde{x_i}) - A^{-1}b| \leqq (z_i p_i)$ の様に求めたい誤差評価を求める工夫を行ってみた．2 の整数乗をかけるのは連立一次方程式の変形時の変形誤差が生じないようにするためである．

3.4.1 数値例

ここでは，提案したアルゴリズムの有効性を示すために行った数値実験を紹介する．5×5 行列で実験をしてみた．MATLAB ではセミコロンで区切れば，一行に幾つもの命令を書くことができる．したがって，`A=10.^(200*random(n));` で条件数の悪い行列 A を生成している．`R=inv(A);` という命令によって，行列 A の逆行列を倍精度浮動小数点数演算で計算している．丸め誤差のため，計算された R は A の近似逆行列になっている．この実験では `norm(A,inf)*norm(R,inf)`, より条件数は 10^{123} 程度である．`b=A*ones(n,1);` としても丸め誤差のため $x = (1, 1, \cdots, 1)^T$ とはならないことに注意した．
`[x,err]=ite_2_4(A,b,10)` の 10 は $R = R_1 + R_2 + \cdots + R_{10}$ の形で変数 10 個を使って R を表現する事を意味する．x が解であり，err がその成分ごと誤差評価である．計算にはおよそ 0.96 秒かかっている．

```
>> mex -O testdot.c
>> mex -O testsumeft.c
>> n=5; A=10.^(200*random(n)); R=inv(A); ...
norm(A,inf)*norm(R,inf), b=A*ones(n,1);
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 3.439096e-124.

ans =

  2.9077e+123

>> format long e
>> tic;[x,err]=ite_2_4(A,b,10),toc
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 3.439096e-124.
```

```
> In C:\MATLAB6p5p1\work\testinv_4.m at line 4
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 3.309116e-169.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 26
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 1.461672e-137.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 26
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 1.460056e-105.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 26
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 4.330720e-072.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 26
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 1.520987e-037.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 26
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5

x =

    1.000000000000000e+000
   -8.588345995753102e+007
    1.636352752301583e+041
    1.000000000000000e+000
   -4.703989110503468e+084


err =

    9.095940312614928e+052
    2.249371489524592e+052
    6.168021609514259e+052
    2.287732647801158e+052
    1.747349865035125e+069
```

```
elapsed_time =

    9.619999999999997e-001
```

この例だと，解が大小ばらばらになったため，絶対値の大きくない解に対する誤差評価がシャープに行えなかった．そこで，解の絶対値をほぼ 1 にそろえる様に A を変形してから誤差評価を行ってみた．この時，A を変形する時の丸め誤差を防ぐため，x の係数にかける数を 2 の整数乗にした．A の条件数はほぼ 10^{74} である．z は変形した方程式の解，$zerr$ は変形した方程式の成分ごと誤差評価，$zerr2$ は x の係数にかけた数を変形した方程式の誤差評価にかけて求めた，変形する前の方程式の成分ごと誤差評価である．変形する前の方程式の誤差評価がシャープに行えるようになった事が分かる．計算時間はおよそ 0.77 秒であった．

```
>> tic;[z,zerr,zerr2]=mDot_4i2_4(A,b,x,err,10),toc
警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 3.416787e-074.
> In C:\MATLAB6p5p1\work\mDot_4i2_4.m at line 7

cnd =

    4.813158365563618e+073

警告: 行列は特異値に近いか，スケールがよくありません.
結果は正しくないかも知れません RCOND = 3.416787e-074.
> In C:\MATLAB6p5p1\work\testinv_4.m at line 4
  In C:\MATLAB6p5p1\work\ite_2_4.m at line 5
  In C:\MATLAB6p5p1\work\mDot_4i2_4.m at line 9

z =

    1.000000000000000e+000
   -1.754830494299174e+000
    2.575746077742598e+000
    1.000000000000000e+000
   -1.660133649004010e+000


zerr =

    5.545365333324048e-032
```

```
    6.407014292725848e-016
    9.404236960781811e-016
    1.058678903026606e-031
    6.061269143223861e-016


zerr2 =

    5.545365333324048e-032
    4.299674508165951e-008
    8.192245790976531e+025
    1.058678903026606e-031
    2.355007802389738e+069


elapsed_time =

    7.710000000000079e-001
```

3.4.2 付録: 精度保証プログラム

実際の精度保証プログラムを示す.

```
function [x,err] = ite_2_4(A,b,K)
system_dependent('setround','nearest');
% inverse of A
R = testinv_4(A,K);    % R = R1 + R2 + ... + Rk

% approximate solution
x = mDot_2_4(R,b,K);

% one residual iteration for backward stability
r = A*x - b;
z = mDot_2_4(R,r,K);
x = x - z;
znormold = norm(z);

% iterative refinement
mloop = 15;
```

```
for loop=1:mloop
    % precise residual
    pres = mDot_2_4([A b],[x; -1],K);
    z = mDot_2_4(R,pres,K);
    znorm = norm(z);
    % update x
    if znorm < znormold
        x = x - z;
    else
        break
    end
    if znorm >= znormold*1e-2, break, end
    znormold = znorm;
end
err = vale_3_4(A,b,x,R,K);
system_dependent('setround','nearest');



function R=testinv_4(A,k)

n=size(A,1);
RR(:,:,1)=inv(A);
for i=2:k
    RR(:,:,i)=zeros(n,n);
end
l=k;

for h=1:k
RR1=RR(:,:,1);
for i=2:l
        RR1=[RR1,RR(:,:,i)];
end
AA=A;
for i=2:l
        AA=[AA;A];
end
for i=1:n
        for j=1:n
```

```
            C(1,1,1:2*n*l)=0;
            C(1,1,:)=testdot(RR1(i,:),AA(:,j)',k);
            CC(i,j)=C(1,1,1);
        end
end
D=inv(CC(:,:,1));
RR2=RR(:,:,1);
for i=2:l
        RR2=[RR2;RR(:,:,i)];
end
DD=D;
for i=2:l
        DD=[DD,D];
end
for i=1:n
        for j=1:n
            RR3(1,1,1:2*n*l)=0;
            RR3(1,1,:)=testdot(DD(i,:),RR2(:,j)',k);
            RR(i,j,1:l)=RR3(1,1,1:l);


        end
end
end
R=RR;



function C = mDot_2_4(A,B,K)
[mA,nA,J] = size(A);
[mB,nB] = size(B);
if nA == mB
    n = nA;
else
    error('Dimensions do not match in mDot_2_4.')
end

AA=A(:,:,1);
for i=2:J
```

```
    AA=[AA,A(:,:,i)];
end
BB=B;
for i=2:J
    BB=[BB;B];
end
for i=1:mA
    for j=1:nB
        CC(1,1,:)=testdot(AA(i,:),BB(:,j)',K);
        C(i,j)=CC(1,1,1);
    end
end



function result = vale_3_4(A,b,x,R,K)
n = length(A);
[Gl,Gu] = mdot_4_3(R,A,K);          % R{1:J} * A - I in J-fold precision
G = max(abs(Gl),abs(Gu));
dd = norm(G,inf);
system_dependent('setround',-inf);
denom = 1 - dd;
if (denom<0)
    disp(denom);
else
    [s,t] = mDot_2i_4([A b],[x; -1],K);
    system_dependent('setround',+inf);
    center = 0.5*(t + s);
    radius = center - s;
    system_dependent('setround',-inf);
    Rs = R(:,:,K)*radius;
    for k=K-1:-1:1
        Rs = Rs + R(:,:,k)*radius;
    end
    system_dependent('setround',+inf);
    Rt = R(:,:,K)*radius;
    for k=K-1:-1:1
        Rt = Rt + R(:,:,k)*radius;
    end
```

```
    system_dependent('setround',+inf);
    newr = abs(R(:,:,1))*radius;
    for k=2:K
        newr = newr + abs(R(:,:,k))*radius;
    end
    system_dependent('setround',-inf);
    Rs = Rs - newr;
    system_dependent('setround',+inf);
    Rt = Rt + newr;
    aa = max(abs(Rs),abs(Rt));
    g = norm(aa,inf)/denom;
    result = aa + g*sum(G')';
end



function [Cl,Cu] = mDot_4_3(A,B,K)
setround(0)
[mA,nA,J] = size(A);
[mB,nB] = size(B);



if nA == mB
    n = nA;
else
    error('Dimensions do not match in mDot_4_3.')
end

AA=A(:,:,1);
for i=2:J
    AA=[AA,A(:,:,i)];
end
BB=B;
for i=2:J
    BB=[BB;B];
end
for i=1:mA
    for j=1:nB
```

```
        CC(1,1,:)=testdot(AA(i,:),BB(:,j)',K);
        if i==j
            CC(1,1,2*n*K+1)=-1;
            CC(1,1,:)=testsumeft(CC(1,1,:),K);
        end
        setround(-1)
        Cl(i,j) = CC(1,1,1)+sum(CC(1,1,2:end));
        setround(+1)
        Cu(i,j) = CC(1,1,1)+sum(CC(1,1,2:end));
        setround(0)
        clear CC;
    end
end



function [Cl,Cu] = mDot_2i_4(A,B,K)
setround(0)
[mA,nA,J] = size(A);
[mB,nB] = size(B);
if nA == mB
    n = nA;
else
    error('Dimensions do not match in mDot_2i_4.')
end

AA=A(:,:,1);
for i=2:J
    AA=[AA,A(:,:,i)];
end
BB=B;
for i=2:J
    BB=[BB;B];
end
for i=1:mA
    for j=1:nB
        CC(1,1,:)=testdot(AA(i,:),BB(:,j)',K);
        setround(-1)
        Cl(i,j) = CC(1,1,1)+sum(CC(1,1,2:end));
```

```
        setround(+1)
        Cu(i,j) = CC(1,1,1)+sum(CC(1,1,2:end));
        setround(0)
    end
end



function [z,zerr,zerr2] = mDot_4i2_4(A,b,x,err,K)
n=size(A,1);
x=floor(log2(abs(x)));
for i=1:n
    A(1:n,i) = A(1:n,i)*(2.^x(i));
end
R = inv(A);
cnd = norm(A,inf)*norm(R,inf)
[z,zerr] = ite_2_4(A,b,K);
setround(+1);
zerr2=zeros(n,1);
for i=1:n
    zerr2(i) = zerr(i)*abs(2.^x(i));
end



function setround(i)
if i==1
    system_dependent('setround',+inf);
elseif i==-1
    system_dependent('setround',-inf);
elseif i==0
    system_dependent('setround','nearest');
end



function A=random(n)
A=2*rand(n)-1;
```

```
/* testdot.c */

#include <stdio.h>
#include <math.h>
#include "mex.h"

#define PP 27.

double tx,ty;
double* ta;

void twosum(double a,double b)
{
double z;

tx=a+b;
z=tx-a;
ty=(a-(tx-z))+(b-z);
}

void split(double a)
{
double c;

c=(pow(2.,PP)+1.)*a;
tx=c-(c-a);
ty=a-tx;
}

void twoproduct(double a,double b)
{
double a1,a2,b1,b2;

split(a);
a1=tx;a2=ty;
split(b);
b1=tx;b2=ty;
```

```
tx=a*b;
ty=a2*b2-(((tx-a1*b1)-a2*b1)-a1*b2);
}

void sumeft(double* a,int n)
{
int i;
double z;

z=a[n-1];
for(i=n-2;i>=0;i--)
    {
    twosum(z,a[i]);
    z=tx;a[i+1]=ty;
    }
a[0]=z;
}

void dot(double* a,double* b,int n)
{
int i;
double p,s;

twoproduct(a[0],b[0]);
p=tx;ta[0]=ty;
for(i=1;i<n;i++)

    {
    twoproduct(a[i],b[i]);



    s=tx;ta[2*i-1]=ty;
    twosum(p,s);
    p=tx;ta[2*i]=ty;
    }
ta[2*n-1]=p;
}
```

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
double* a;
double* b;
int k;
int i,n;

n=mxGetN(prhs[0]);
a=mxGetPr(prhs[0]);
b=mxGetPr(prhs[1]);
plhs[0]=mxCreateDoubleMatrix(1,2*n,mxREAL);
ta=mxGetPr(plhs[0]);
dot(a,b,n);
k=mxGetScalar(prhs[2]);
for(i=0;i<k;i++)
    sumeft(ta,2*n);
}



/* testsumeft.c */

#include <stdio.h>
#include "mex.h"

double tx,ty;
double* ta;

void twosum(double a,double b)
{
double z;

tx=a+b;
z=tx-a;
ty=(a-(tx-z))+(b-z);
}

void sumeft(double* a,int n)
```

```
{
int i;
double z;

z=a[n-1];
for(i=n-2;i>=0;i--)
    {
    twosum(z,a[i]);
    z=tx;a[i+1]=ty;
    }
a[0]=z;
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
double* a;
int k;
int i,n;

n=mxGetN(prhs[0]);
a=mxGetPr(prhs[0]);
k=mxGetScalar(prhs[1]);
for(i=0;i<k;i++)
    sumeft(a,n);
plhs[0]=mxCreateDoubleMatrix(1,n,mxREAL);
ta=mxGetPr(plhs[0]);
for(i=0;i<n;i++)
    ta[i]=a[i];
}
```

第4章　結論

本論文では連立一次方程式

$$Ax = b$$

に対する数値解 $\tilde{x}$ の精度保証付き数値計算法について考えた．残差反復法を

1. 残差 $\tilde{r} = A\tilde{x} - b$ を高精度に計算する．
2. 連立一次方程式 $Ap = \tilde{r}$ を解く．
3. 近似解 $\tilde{x}$ を $\tilde{x}_{new} \leftarrow \tilde{x} - p$ と更新する．

の様に所望の精度が得られる回数行って得られた精度の高い計算解に対する精度保証をシャープに行う方法を提案し，さらに係数行列の条件数 $\|A\|\|A^{-1}\|$ が $\mathcal{O}(10^{16})$ 以上になっても A の逆行列を R，単位行列を I として

$$\|RA - I\|_\infty < 1$$

となる位まで高精度に逆行列を求め精度保証がシャープに出来るようにした．

第 1 章では第 2 章と第 3 章の準備として高精度内積計算アルゴリズムと連立 1 次方程式の精度保証法を紹介した．

第 2 章では係数行列の条件数が $\mathcal{O}(10^{16})$ 以下の時に Ogita, Rump, Oishi による高速な任意精度内積演算アルゴリズムを用いることによって高速性を損なう事無く高精度に精度保証を行えるようにした．これにより残差反復法によって得られた精度の高い計算解をシャープに精度保証をする事が出来るようになった．さらに，条件数が $\mathcal{O}(10^{16})$ に近い場合と行列の次元が大きい場合で数値実験を行い高速性を損なう事無く高精度に精度保証を行え，残差反復法による精度の高い計算解をシャープに精度保証を行える事を示した．しかし，条件数が $\mathcal{O}(10^{16})$ 以上になるとこの精度保証法は逆行列の精度が足りないために適用出来なくなってしまう．

そこで，第 3 章では，係数行列の条件数が非常に大きい場合の連立一次方程式に対する精度保証付き数値計算法を提案した．条件数が非常に大きいという事は行列が非正則に近くて計算誤差が出やすいという事であるが，非正則でなければどんなに条件数が大きくても精度保証が出来るようになった．ここでは，残差反復法を逆行列の計算に応用して逆行列を倍精度浮動小数点数の行列の和の形で表現する事によって逆行列を高精度に求め条件数が大きい係数行列の連立一次方程式に対しても精度保証が出来るようにした．本提案方式は，基本的に内積計算の高精度演算が可能であれば IEEE754 規格に従う計算機上で実装可能である．内積計算の高精度演算は比較的高速なものが開発されてきていて，全ての演算を多倍長浮動小数点数

演算で計算する場合と比べて内積だけを高精度に計算する方が圧倒的に速くなる．数値実験によって，条件数が非常に大きい場合でも，また係数行列の次元数が大きめでも，所望の精度の近似解が精度保証付きで計算されることを示した．残差反復法により精度の高い計算解が得られ，その精度をシャープに評価出来ていることが分かる．また，多倍長浮動小数点数演算を用いても悪条件性を解決する事が出来るが，この精度保証にかかる計算時間は多倍長浮動小数点数演算と比べてかなり高速にする事に成功した．数値実験の結果からも多倍長浮動小数点数演算よりも行列積と行列ベクトル積のみを高精度に計算する提案方式の方が 10 倍以上早いことが示された．さらに，多倍長浮動小数点数演算では逆行列の精度が足りないと逆行列の計算を始めからやり直さねばならなかったが，提案方式では事前に逆行列の計算に必要な精度が分からなくても反復計算により問題に応じて精度を増やしながら必要なだけの計算を行えるようにした．ただし，オーバフローが起きるような，数値が倍精度の範囲を越えるような問題などでは，多倍長演算が必要になる場合もある．

第 3.4 節では計算解の絶対値の大小が極端に指数関数的にばらばらになった時には Yamamoto の定理を用いても絶対値の小さな解に対する精度保証が絶対値の大きな成分の影響でシャープに行えなかったものがスケーリングを行い絶対値の差が小さい方程式に置き換えて計算して元の方程式での精度保証に換算する事によって絶対値の小さな解に対する精度保証がシャープに行えるようにした．この場合，スケーリングを行う前は残差反復法による高い精度の計算解がシャープに評価出来ていなかったものが，スケーリングを行うことによりシャープに評価出来る様になっていることが分かる．計算時間はシャープに評価出来ない事を確認した後スケーリングを行って精度保証をやり直すので，スケーリングを行わなくて済む場合と比べて倍以上かかる事になる．Yamamoto の定理を用いても絶対値の小さな解に対する精度保証がシャープに行えないという問題は条件数の大きい係数行列での数値実験をしていて初めて出あった問題である．

どのような条件下で近似逆行列の収束性が保証されるのか等についての証明は，今後の研究課題である．

謝辞

本研究を進めるにあたり，ご指導ならびにご鞭撻を賜り暖かくご激励いただきました指導教授の大石進一先生に心より感謝いたします.

並びに，本研究を進める過程で様々な議論をしていただくなど，多大なご助力を賜わりました早稲田大学理工学術院客員教授の田邉國士先生，同教授の高橋大輔先生及び同助教授の柏木雅英先生に心より感謝いたします.

また，共同研究者の科学技術振興機構 · 早稲田大学理工学術院客員講師の荻田武史先生，ハンブルク工科大学教授の Siegfried M. Rump 先生から多大なるご指導，ご助言をいただいた事に対して大変深く感謝しております.

最後に，研究において共に協力し，ご激励いただきました大石研究室のメンバー諸氏に感謝いたします.

参考文献

[1] ANSI/IEEE: *IEEE Standard for Binary Floating Point Arithmetic*, Std 754–1985 ed., New York, IEEE, 1985.

[2] T. J. Dekker: A floating-point technique for extending the available precision, *Numer. Math.* 18 (1971), 224–242.

[3] N. J. Higham: Accuracy and Stability of Numerical Algorithms, 2nd ed., SIAM Publications, Philadelphia, PA, 2002.

[4] R. B. Kearfott, M. Neher, S. Oishi, F. Rico: Libraries, tools, and interactive systems for verified computations: four case studies, in Numerical Software with Result Verification (R. Alt, A. Frommer, R. B. Kearfott, and W. Luther eds.), *Lecture Notes in Computer Science*, 2991, Springer-Verlag, Heidelberg, 2004.

[5] D. E. Knuth: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Reading, Massachusetts: Addison-Wesley, 1969.

[6] U. W. Kulisch, W. L. Miranker: The arithmetic of the digital computer: A new approach, *SIAM Review*, 28 (1986), 1–40.

[7] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo: Design, implementation and testing of extended and mixed precision BLAS, *ACM Trans. Math. Softw.*, 28 (2002), 152–205.

[8] T. Ogita, S. Oishi, Y. Ushiro: Computation of sharp rigorous componentwise error bounds for the approximate solutions of systems of linear equations, *Reliable Computing*, 9:3 (2003), 229–239.

[9] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot Product, *SIAM J. Sci. Comput.*, 26:6 (2005), 1955-1988.

[10] S. Oishi, S. M. Rump: Fast verification of solutions of matrix equations, *Numer. Math.*, 90:4 (2002), 755–773.

[11] D. M. Priest: On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations, PhD thesis, University of California at Berkeley, USA, 1992.

[12] S. M. Rump: Approximate inverses of almost singular matrices still contain useful information, Forschungsschwerpunktes Informations- und Kommunikationstechnik, Technical Report 90.1, Technical University Hamburg-Harburg, 1990.

[13] S. M. Rump: A class of arbitrarily ill-conditioned floating-point matrices, *SIAM J. Matrix Anal. Appl.*, 12:4 (1991), 645–653.

[14] T. Yamamoto: Error bounds for approximate solutions of systems of equations, *Japan J. Appl. Math.*, 1:1 (1984), 157–171.

[15] 太田 貴久, 荻田 武史, S. M. Rump, 大石 進一: 悪条件連立一次方程式の精度保証付き数値計算法, 日本応用数理学会論文誌, 15:3 (2005), 269–287.

[16] 荻田 武史, 大石 進一: 大規模連立一次方程式のための高速精度保証法, 情報処理学会論文誌: 数理モデル化と応用, 46:SIG10 (TOM12) (2005), 10–18.

著者の関連業績

1. 太田 貴久, 大石 進一, 荻田 武史, S. M. Rump: “高精度内積計算アルゴリズムを用いた連立一次方程式の精度保証付き数値計算法”, 日本シミュレーション学会大会, 2004 年 6 月, 発表論文集 pp. 345–348.

2. 太田 貴久, 荻田 武史, S. M. Rump, 大石 進一: “条件数の非常に大きい連立一次方程式の精度保証付き数値計算法”, 平成 17 年春の応用数理学会研究部会・同準備会連合発表会, 2005 年 3 月.

3. 太田 貴久, 荻田 武史, 大石 進一, S. M. Rump: “悪条件連立一次方程式の精度保証付き数値計算法”, 計算機援用証明チュートリアル 2005, 2005 年 3 月.

4. 太田 貴久, 荻田 武史, S. M. Rump, 大石 進一: “条件数が非常に大きい連立一次方程式に対する解の精度保証法”, 日本シミュレーション学会大会, 2005 年 7 月, 発表論文集 pp. 225–228.

5. 太田 貴久, 荻田 武史, S. M. Rump, 大石 進一: “悪条件連立一次方程式の精度保証付き数値計算法”, 日本応用数理学会論文誌, 15:3 (2005), pp. 269–286.

6. T. Ohta, T. Ogita, S. M. Rump and S. Oishi: “Numerical verification method for dense linear systems with arbitrarily ill-conditioned matrices”, Proceedings of 2005 International Symposium on Nonlinear Theory and its Applications, Bruges, Belgium, 2005, pp. 745–748.